Stanford's CS231n Convolutional Neural Networks for Visual Recognition (Spring 2017)

Sriram Sami

February 2, 2020

Contents

1 Lecture 1: Big Picture

1.1 Motivation

Estimated that $> 85\%$ of data online is "pixel-data". So image data is like the "dark mattaer" of the web - tons of it out there that sits, un-nalyzed.

1.2 Visual cortex structure

We ourselves mostly visualize objects first as simple edge-like features. So when we see neural nets do the same thing, it seems like a deep result. We believe vision processing to be **hierarchical**.

1.3 History

Edge detection → Objects are compositions of basic shapes when viewed from a particular angle → Normalized cut as an attempt to group things into objects → decision making in vision by engineering important **features** about the object → PASCAL standardized image recognition **datasets** for competing on these tasks → ImageNetd dataset from Stanford
Point of the course *Image classification* - what is in whole image X?

2 Lecture 2: Image Classification Pipeline

2.1 Input

An image is just a large matrix of numbers to a computer. Imagine one sub-matrix for each channel R, G, B, that's 800 x 600, and each value is between 0 - 255 (intensity). (e.g. 800 x 600 x 3).

Challenges:

1. Viewpoint variation: moving the camera will change the entire pixel grid
2. Illumination conditions: changes intensity values
3. Deformation: Shape changes for object
4. Occlusion: only see a small portion of object
5. Background clutter: looks similar to BG
6. Intraclass variation: many different "types" of object X

Difficulty is that **you can't sit down and write an explicit algorithm** to classify these images.

2.2 Data Driven Approach

The core idea that we should **train** the model with lots of training images and known labels, which spits out a **model**. Use **model** in production to actual recognize unknown images.

2.3 Simple Classifier: Nearest Neighbor

1. Training step: do nothing except memorize all training data (store in some representation).
2. Prediction step: take new image and find the most similar image in the training set, predict that class.

Note: lecture shows the short code for this algo.

2.3.1 How to compare two images?

L1 distance (Manhattan): compare individual pixels between two images, take abs, sum across all pixels. One dumbish way to do this. Where p is a point/pixel in the image: $d_1(I_1, I_2) = \sum_p(\mid I_1^p - I_2^p \mid)$

L2 distance (Euclidean): Sqrt of the sum of the squares of the (difference between each two points). Where p is a point/pixel in the image: $d_2(I_1, I_2) = \sqrt{\sum_p(I_1^p - I_2^p)^2}$

2.3.2 How fast?

Train O(1), predict O(N). Really slow. We want classifiers to be slow to train but fast to test by comparison.

2.3.3 Small adjustment: k-nearest neighbors

Don't just look for the nearest neighbor: check the k-nearest neighbors, and "take a vote" based on these neighbors. Generally a **majority** vote.

2.3.4 Limitations

Issue: L1 and L2 distance not so good at handling small perceptual differences for images: shifting image left / right, redactions, tints... Argument is that the distances can be made arbitrarily similar even though the actual two images are very perceptually different.

Curse of dimensionality: Your nearest neighbors need to be close by to the test image to have confidence in the classification. Bad growth with image size: need a *lot more training data* to densely cover space.

2.4 Hyperparameters

Choices about the algorithm (not about the data itself) that you have to make. E.g., which distance metric, what value of K.

2.4.1 Choosing Hyperparameters

Best plan: separate data into THREE separate partitions

1. Training set: (most of data) - train algorithm with many different hyperparameters choices on training set.
2. Validation set: use results of model on the validation set to choose best hyperparameters.
3. Test set: when everything is **done**, run **once** on your test set. Report that number. **You want to know how your algorithm performs on unseen data!**

2.5 Cross validation

Partition training data into many "folds". Try each fold as a validation set. Average results across all folds. This is mostly used in smaller datasets are not deep learning. **This is more of a gold standard for generalizing your hyperparameters**, but this is very computationally expensive.

2.6 Linear Classification Algorithm

Linear classifier is an **example of a *parametric model***.

2.6.1 General Parametric Model

Image (x) + Parameters (W) $\rightarrow f(x, W) \rightarrow$ a score for each of the possible classes.

I.e. we **do not keep the training images** during the **test phase**. We **summarize them** in the set of **parameters** W after training.

Difficulty is in **choosing structure for function f**.

2.6.2 Linear Classifier

Just **multiply weights and inputs**.

$f(x, W) = Wx + b$.

Assuming input image is 32 x 32 x 3, so take x as a column vector of $32x32x3 = 3072$ values. To get 10 classes for output (10 x 1 vector), W must have dimension 10 x 3072. Each weight row corresponds to weights for that class.

b is a constant bias vector of $10x1$ in this case which is some class-independant data (example given is if dataset is unbalanced wit more cats than dogs, bias element for cats would be higher than dogs).

A nice property: can **visualize the rows in the weight matrix** to see what kind of images the classifier has learnt! I.e. what the weights for that category are most sensitive to.

Problem: **linear classifier can only learn one template for each class**. I.e, learnt weights show a horse-like image with two heads (since it could be oriented either way), etc.

Linear classifier is just learning **linear decision boundaries** (lines / planes / hyperplanes) to separate all classes. Lecture at 56 mins has great example of datasets where this decision region boundary fails miserably.
Figuring out how to get the correct weight matrix: next lecture.